UNIVERSITY of York

This is a repository copy of *Development Automation of Real-Time Java: Model-Driven Transformation and Synthesis*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/159681/

Version: Accepted Version

Article:
Chang, Wanli orcid.org/0000-0002-4053-8898, Wei, Ran, Zhao, Shuai et al. (3 more authors) (Accepted: 2020) Development Automation of Real-Time Java: Model-Driven Transformation and Synthesis. ACM Transactions in Embedded Computing Systems. ISSN 1558-3465 (In Press)

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Development Automation of Real-Time Java: Model-Driven Transformation and Synthesis

WANLI CHANG, Department of Computer Science, University of York, UK
RAN WEI, School of Artificial Intelligence, Dalian University of Technology, China
SHUAI ZHAO, Department of Computer Science, University of York, UK
ANDY WELLINGS, Department of Computer Science, University of York, UK
JIM WOODCOCK, Department of Computer Science, University of York, UK
ALAN BURNS, Department of Computer Science, University of York, UK

Many applications in emerging scenarios, such as autonomous vehicles, intelligent robots, and industrial automation, are safety-critical with strict timing requirements. However, the development of real-time systems is error-prone and highly dependent on sophisticated domain expertise, making it a costly process. This paper utilises the principles of model-driven engineering (MDE) and proposes two methodologies to automate the development of real-time Java applications. The first one automatically converts standard time-sharing Java applications to real-time Java applications, using a series of transformations. It is in line with the observed industrial trend, such as for the big data technology, of redeveloping existing software without the real-time notion to realise the real-time features. The second one allows users to automatically generate real-time Java application templates with a light-weight modelling language, which can be used to define the real-time properties — essentially a synthesis process. This paper opens up a new research direction on development automation of real-time programming languages and inspires many research questions that can be jointly investigated by the embedded systems, programming languages as well as MDE communities.

CCS Concepts: • **Computer systems organization → Real-time systems**; • **Software and its engineering → Software notations and tools**.

Additional Key Words and Phrases: Real-Time Programming Languages, Real-Time Specification for Java, Model-Driven Engineering

ACM Reference Format:
Wanli Chang, Ran Wei, Shuai Zhao, Andy Wellings, Jim Woodcock, and Alan Burns. 2020. Development Automation of Real-Time Java: Model-Driven Transformation and Synthesis. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (April 2020), 26 pages. https://doi.org/ab.cdef/1234567.7654321

Authors' addresses: Wanli Chang, wanli.chang@york.ac.uk, Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK; Ran Wei, ranwei@dlut.edu.cn, School of Artificial Intelligence, Dalian University of Technology, Dalian, China; Shuai Zhao, shuai.zhao@york.ac.uk, Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK; Andy Wellings, andy.wellings@york.ac.uk, Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK; Jim Woodcock, jim.woodcock@york.ac.uk, Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK; Alan Burns, alan.burns@york.ac.uk, Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
1539-9087/2020/4-ART1 $15.00
https://doi.org/ab.cdef/1234567.7654321

1 INTRODUCTION

Stringent temporal requirements are widely encountered in emerging scenarios like autonomous vehicles, intelligent robots, and industrial automation that support safety-critical applications. A *real-time system* must react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment [10]. In [29], the author classifies system failure modes into *random failures* and *systematic failures*, the latter contributing to hazards that could lead to incidents with catastrophic consequences. *Systematic failures* can be further classified into *functional failures* and *timing failures*. It is imperative to ensure that a safety-critical system imposes correct timing requirements and at the same time, that such requirements are satisfied by the system's timing behaviour. Demonstrating real-time properties forms key evidence in certifying the safety of a system.

Due to the high productivity, portability and relatively low maintenance cost, the *Java* programming language has received extensive attention in the real-time and safety-critical domains [24, 54]. For instance, Java was adopted in [35] and [34] to reduce distributed computing latency in an unified cloud-based platform for autonomous vehicles. However, these works have been developed focusing on functionality with limited consideration of timing and safety guarantee, especially when the complex perception functions are involved. As mandated by safety regulations, such as the ISO 26262 for automotive systems and IEC 61508 for functional safety, hard real-time constraints are essential to guarantee safety of the system (e.g., the vehicle) and its surrounding environment. Thus, there is a need to push these existing works towards the real-time regime.

There is a tendency that matured Java techniques (which were developed without the notion of real-time) are re-developed to possess real-time properties (e.g., real-time big data systems [21] and real-time stream processing techniques [37]). The major reason is that, those simple and conservative methods (like leaving large safety margins) that were deployed in practice are losing ground, with ever more complicated functionality, higher timing-related performance requirements and limited resources on the emerging real-time applications [3, 14–16].

Despite its popularity, standard Java does not provide real-time related facilities such as thread scheduling, resource sharing control, memory management, etc., which are essential to achieve predictability [11] in terms of temporal behaviour. This has motivated the development of the *Real-Time Specification for Java* (RTSJ) [8]. RTSJ reserves intrinsic advantages of Java, and provides plenty of real-time facilities to guarantee system temporal behaviour. However, RTSJ is arguably harder to use than standard Java due to its rather complex real-time facilities.

Compared to generic time-sharing applications in Java, developing real-time applications using RTSJ depends greatly on the level of expertise in real-time systems design, and requires thorough understanding of its specification. Developing with RTSJ is also an error-prone task due to the complexity of the source code. All of the above make the development of real-time applications a costly process. In addition, although there have been system analysis and verification techniques [40] to ensure correctness in the design phase, in terms of both logical and temporal behaviour, it remains an open and challenging problem of how to eliminate human-related erroneous factors (e.g., caused by limited understanding of the real-time concepts and insufficient experience with RTSJ facilities). The safety-critical nature in many real-time application domains amplifies the impact of such concerns.

Model-driven engineering (MDE) is a contemporary software development paradigm, which promotes *models* as first-class artefacts. Based on models, developers are able to perform a series of model management operations in an automated manner, and eventually produce software artefacts, such as documentation and working code. This reduces the amount of time required to develop a system and thus improves the productivity of software engineers, by at least a factor of 10 in

many cases [26, 28]. Adopting MDE also reduces the number of errors throughout the development process and improves consistency [63]. In addition, MDE can be applied to any domain to achieve automation, due to the concept of domain-specific modelling and the interoperability provided by the model management operations, which can be executed in an automated manner.

An initial attempt to automate the development of real-time Java applications based on MDE has been made in [17], which proposes a toolchain that transfers standard Java applications into real-time ones by RTSJ. However, transformation is only provided from certain basic elements in Java (e.g., Java threads to RTSJ threads), and it is not a comprehensive approach for automating the transformation of the entire run-time environment. In addition, it does not consider the scenario where a standard Java reference implementation is not available for such a transformation.

In this paper, we apply the principles of MDE in the domain of real-time programming with Java. We propose two methodologies to automate the development of real-time Java applications. Note that the term "real-time Java" indicates real-time systems (i.e., systems with strict temporal requirements) that are developed by Java semantics (e.g., RTSJ). The first methodology is able to automatically convert existing time-sharing Java applications to real-time applications in RTSJ, through a series of model management operations. The output software is in full compliance to the RTSJ specification, with dependencies on the RTSJ runtime environment supporting scheduling, memory management, resource sharing, asynchrony, etc. In comparison to [17], asynchronous event handling as a major component in Java is included, and asynchronous transfer of control is enabled to make a comprehensive automated transformation of the entire run-time environment. The second one allows users to directly generate a RTSJ-compliant application template with a light-weight modelling language, where users can define customised real-time properties (e.g., scheduling parameters and resource sharing methods) for their objects in these RTSJ models. This is essentially a synthesis process, and especially valuable when a standard Java reference implementation is not available (e.g., to develop a new RTSJ application). The automation toolchain and modelling process associated with the proposed methodologies are explained in Sections 4 and 5, respectively.

These two methodologies enable the developers with limited real-time background to develop real-time applications with high software development efficiency. The output real-time application is verified automatically by the applied methods via the predefined analysis (e.g., the temporal analysis). Due to the application of MDE techniques, the productivity and consistency are significantly improved throughout the development process. In addition, human-related errors (e.g., implementation mistakes) are eliminated in the automation. The scientific challenges addressed and hidden issues discovered towards automatic generation of real-time applications with MDE techniques are also discussed. Furthermore, We identify and describe several future research directions beyond this paper.

The rest of the paper is organised as follows. Section 2 provides a review of the MDE technology and its application in the real-time systems development. Section 3 describes the real-time Java, The proposed Java to RTSJ transformation methodology and its associated toolchain are reported in Section 4 with detailed transformation approaches. Section 5 presents the proposed RTSJ application synthesis methodology via a light-weight modelling language, which provides an alternative when the standard Java source code is not available as input. Section 6 outlines open questions and possible research directions that are introduced by the proposed methodologies. Finally, Section 7 gives the conclusion.

2 MODEL-DRIVEN ENGINEERING

Modelling is an essential part of any system engineering process. Engineers of all disciplines construct models of the systems they intend to build to capture, test and validate their system design ideas with other stakeholders before committing to a long and costly production process.

MDE is a software engineering methodology that aims to reduce the complexity of software systems by promoting *model*s that focus on the essential complexity of systems, as the first-class artefacts of the software development process. In contrast to traditional software development methodologies, where models are mainly used for communication and post-mortem documentation process (e.g. with UML diagrams), in MDE, models are the main living and evolving artefacts from which concrete software development artefacts can be produced in an analysable and automated fashion.

MDE was proposed at the time when object-oriented techniques reached a point of exhaustion [7, 44]. MDE constitutes the latest paradigm shift in software engineering as it raises the level of abstraction beyond that provided by 3rd generation programming languages. In recent studies, MDE has been shown to increase the productivity of developers by as much as a factor of 10 [26, 28], and significantly enhance important aspects of the software development process such as maintainability, consistency and traceability [38].

There are two important aspects of MDE — (i) *domain-specific modelling*, where domain experts create their own domain-specific modelling languages (DSMLs) to capture the concepts in their domain (and create instances of their DSMLs to model their systems), without concerns of lower-level implementation details (such as what programming language and what types of database to use, etc.); (ii) *model management operations*, which are programs performed on models in an automated manner to generate software engineering artefacts. Model management operations typically include, but are not limited to:

- Text-to-Model Transformation (T2M): to convert text (such as source code) into models based on parsing rules defined in the transformation;
- Model Validation: to check the well-formedness of models, as well as custom constraints against the elements in models;
- Model-to-Model Transformation (M2M): to interoperate between different modelling technologies, where one type of model is transformed into another type;
- Model-to-Text Transformation (M2T): to generate text based on the contents of the model (e.g., documentation generation and source code generation);
- Model Comparison: to compare different versions of a model to find out what is changed;
- Model Merging: to integrate models defined by different parties but share model elements.

There are a wide range of tools to create domain-specific modelling languages (DSMLs), such as Eclipse Modeling Framework (EMF) [51], Eclipse-based UML tools [33], MagicDraw [36] and SySML [52]. Amongst these tools, EMF has been widely adopted for its easy-to-use Ecore metamodel, with which the developers are able to create their DSMLs in UML-like diagrams, and may use the created models to generate Java source code. An example Flowchart metamodel created using EMF is shown in Listing 1. With the Flowchart metamodel, developers are able to create instances of the metamodel (namely Flowchart models). An example flowchart model is shown in Figure 1.

```
@namespace(uri="flowchart", prefix="flowchart")
package flowchart;
abstract class NamedElement {
   attr String[1] name;
}
class Flowchart extends NamedElement {
```

```
   val Node[*] nodes;
   val Transition[*] transitions;
}
abstract class Node extends NamedElement {
   ref Transition[*]#source outgoing;
   ref Transition[*]#target incoming;
}
class Transition extends NamedElement {
   ref Node[1]#outgoing source;
   ref Node[1]#incoming target;
}
class Action extends Node { }
class Decision extends Node { }
class Subflow extends Flowchart, Node { }
```

Listing 1. Example Flowchart metamodel

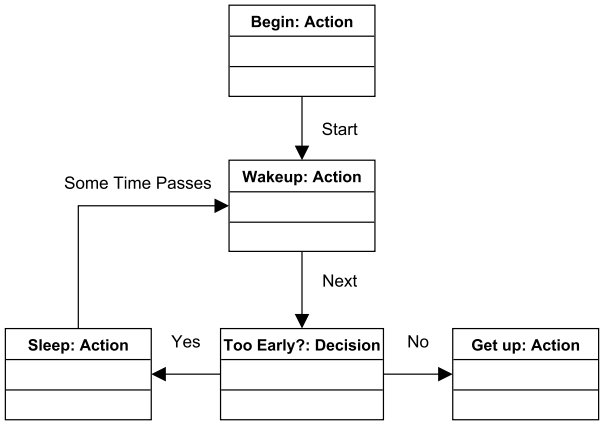

Fig. 1. An example Flowchart model

In EMF, models are XML-based documents which can be consumed by model management programs. In Listing 2, an example model-to-text transformation program is presented, which can be used to create HTML documents based on the Flowchart model in Figure 1. In this way, the development time for these HTML documents can be significantly reduced due to the automation introduced by the model transformation. It is to be noted that due to the high flexibility of model-to-text transformation, the target document can be of any type, including source code written in any programming language.

```
[% import 'util.eol'; %]
<h1>[%=action.name%]</h1>
<br/>
[%var nextSteps = action.outgoing.collect(t : Transition | t.target);%]
[% if (not nextSteps.isEmpty()) { %]
<a href="[%=nextSteps.first().name.clean()%].html">Next step</a>
[% } else { %]
<b>finished</b>
[% } %]
```

Listing 2. Example model-to-text transformation to generate HTML documents

MDE has been applied to a variety of domains with proven benefits. In [31] MDE is applied to transform model query languages to MySQL queries to reduce the effort and error rates in manually creating MySQL queries. In [63], MDE is applied to automatically generate fully functional graphical editors for UML profiles. In [5], MDE is applied to transform natural languages to database query languages to form complex query using simple natural language grammars. In [57], MDE is considered to be the foundation of run-time safety assurance of Cyber-Physical Systems.

Developing real-time systems using model-based approaches has been explored in the community [27, 55]. However, none of these works study the migration from standard Java to real-time Java. In addition, many of the past efforts rely on the notion of model-driven architecture, which is an outdated model based practice and lacks tool support. By applying MDE techniques, as previously described, real-time system developers can benefit from the productivity gain from MDE, as well as the consistency and maintainability through automation provided by MDE.

3 REAL-TIME SPECIFICATION FOR JAVA

The Real-Time Specification for Java (RTSJ) was originally developed as Java Special Request 1 under the Java Community Process in 2001[1]. Since then, RTSJ has been well practised in a wide range of application domains, including automotive, manufacturing control, avionics and information systems [25, 50, 55, 56]. For instance, RTSJ has been applied to the auto-pilot system of an unmanned aerial vehicle, which is the first Java-based system that meets all Boeing's operational requirements for test flights [1]. *Jcoap*, implemented using RTSJ, provides real-time communications for IoT systems [32]. In [20], RTSJ has been applied in a real-time big data processing systems with FPGA-based hardware acceleration. In industry, *JamaicaCAR* developed by both Acis and Perrone Robotics[2] provides a lightweight application framework for car headunits and in-vehicle information systems. In addition, Acis and CLAAS[3] present solutions (namely *Jamaica-IoT*) for digital factory and manufacturing, which enables deployment and operation of data analytics and control logic at the network's edge.

The RTSJ is designed to support both hard and soft real-time applications. This specification consists of two major components — (i) extensions from the Java programming language; and (ii) modifications on the semantics of the standard Java Virtual Machines (JVM) [8]. In this section, we briefly review the programming specification of RTSJ, together with its reference implementations as well as the supporting Virtual Machines (VM). Detailed descriptions of each RTSJ facility and the application examples can be found in [10] and [58].

3.1 Programming Specification

There are several major extensions to the standard Java language that are provided in the package `javax.realtime`, including task scheduling and dispatching, memory management, shared resource control, asynchronous event handling, etc.

One major facility provided in RTSJ is`javax.RealtimeThread`, which takes a set of scheduling-related parameters (e.g., priority, period and deadline) specifying a real-time thread's release, execution and timing properties. Three types of threads are derived from this entity: periodic, sporadic and aperiodic, depending on the input release parameter assigned to the thread's constructor

[1]https://jcp.org/en/jsr/detail?id=1
[2]https://www.perronerobotics.com
[3]https://www.claas.ca

method. In addition, a set of asynchronous event handlers are provided to allow user-defined actions in the cases of deadline miss or budge overrun. By default, the real-time threads are scheduled by a preemptive fixed-priority scheduler, but user-defined scheduling and dispatching policies are also possible.

Another important extension is the real-time memory management model. In RTSJ, a set of memory management facilities are provided in RTSJ (e.g.,`ImmortalMemory` and `ScopedMemory`) to allow the construction of self-defined memory models. However, RTSJ imposes a set of memory accessing rules that restrict memory-accessing behaviours to prevent *dangling reference* (i.e., references that point to objects in reclaimed memory blocks). With memory management model defined, the standard Java garbage collector is no longer required so that its unpredictable interference is avoided at run-time. Later, a real-time garbage collector is provided by JamaicaVM (see Section 3.2), which allows the use of Heap memory and eases the development of RTSJ applications by avoiding building complex memory models.

In the presence of shared objects, RTSJ provides several resource sharing policies like priority Inheritance [47] and Priority Ceiling Protocol (PCP) [41]. Among these protocols, the PCP yields minimal blocking time (i.e., one critical section only) and guarantee dead-lock free resource accesses. In addition, asynchrony is well handled via a set of asynchronous event handling facilities. Finally, a set of time-related facilities (e.g., real-time system clock and `HighResolutionTime` with granularity of nanoseconds) are supported.

3.2 RTSJ Implementations and VMs

RTSJ was firstly implemented by TimeSys[4]. This implementation (i.e., a RTSJ-compliant VM and a RTSJ reference implementation) supports all versions of Linux. Later, Aicas GmbH[5] provided a different RTSJ implementation in their JamaicaVM, supporting a wide range of real-time operating systems, such as Linux, VxWorks and QNX. In addition, there are also other virtual machines which are compliant with RTSJ, e.g., jRate[6], OVM [4] and Aero JVM[7].

Among these VMs, JamaicaVM provides hard real-time guarantees and is the mostly adopted VM for executing RTSJ applications. Currently, JamaicaVM supports RTSJ V1.0.2 and is working towards RTSJ 2.0 implementation based on Java 8. In particular, a real-time Garbage Collector (GC) is supported by JamaicaVM [48]. This GC executes each time when threads issues requests to allocate an object in a preemptable fashion, and will not interrupt application threads. In JamaicaVM manual[8], an analytical approach for measuring worst-case execution time in the presence of the real-time garbage collector is provided.

In total, thirty-eight priority levels are supported by this VM, where priority levels 11-38 are designated for real-time threads (through the class `RealtimeThread`) and priority levels 1-10 are designated to standard Java threads. That means JamaicaVM is also compliant with the standard non real-time threads. In this work, we assume that each thread in the given application will be mapped to a real-time thread, and each real-time thread has a unique priority.

3.3 Targeted RTSJ Run-Time Environment

In testing the feasibility of our approach, we assume a simple but widely applied real-time system model. Standard Java applications in uniprocessor systems are transformed to real-time applications, where Java threads can be converted to either periodic or aperiodic real-time threads, with their

[4]https://www.timesys.com
[5]https://www.aicas.com/cms/en
[6]http://jrate.sourceforge.net/
[7]http://www.aero-project.org
[8]https://www.aicas.com/cms/sites/default/files/JamaicaVM-8.2-manual-web.pdf

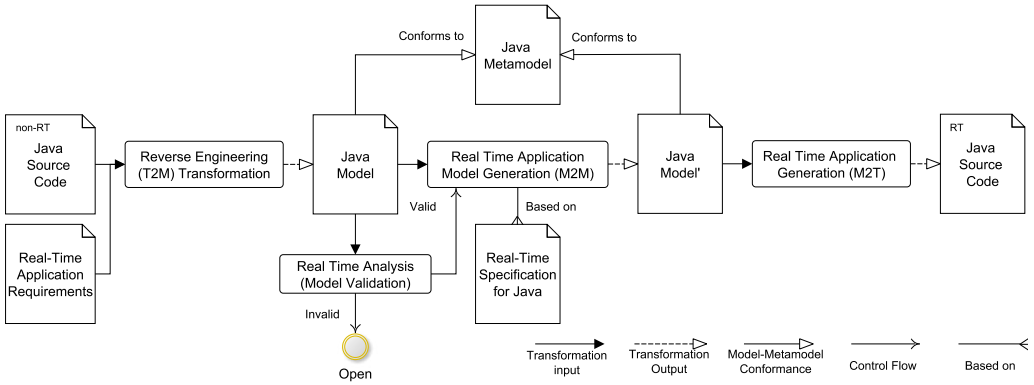

Fig. 2. Time-sharing applications to real-time applications migration

release parameters pre-defined in the application requirements. The Fixed-Priority Preemptive Scheduling (FPPS) policy is taken for coordinating executions of real-time threads due to its being widely applied and well-supported exact schedulability test. The scheduling policy is set by the underlying real-time operating system (e.g., RTEMS [43]) and not configurable by the proposed toolchain. Other scheduling policies can be used, as long as they have schedulability tests in place, such as the Earliest Deadline First scheduling with the Processor Demand Analysis [12].

In addition, threads can access shared objects, but they must do so with the PCP applied, which is an optimal resource sharing solution in uni-processor systems (i.e., deadlock-free and minimised blocking time) [18]. JamaicaVM v8.5 (with RTSJ v1.0.2 and Java 1.5) is used as the underlying virtual machine. Finally, Memory management is handled by the real-time GC provided by JamaicaVM.

4 JAVA TO REAL-TIME JAVA TRANSFORMATION

The structure of our first proposed methodology is shown in Figure 2. The first step in our approach is the reverse engineering of the Java programs into models. In order to do this, we use a Text-to-Model transformation to convert the source code of standard Java applications (i.e. without real-time properties) into Java models. In addition to the Java source code, we also take a list of real-time application requirements that provide necessary information, e.g., the worst-case execution time (WCET), period, priority, and deadline of each thread, for building a real-time system. To ensure the temporal requirements, the WCET of each task provided by the users should be either measured or analysed considering the underlying hardware.

With the two inputs, a Java model that conforms to the Java metamodel is produced. With the Java model, there is a need to perform a model validation to check if the given application is capable to satisfy all the temporal requirement after being transformed to a real-time application. If the model validation passes (the response time of each real-time thread is equal to or less than its deadline), it means that the to be transformed application is schedulable. We then perform a model-to-model transformation to transform the Java model to the target Java model (named Java Model′) which uses RTSJ Java constructs. This transformation is a endogenous transformation - that the target model also conforms to the Java Metamodel (for RTSJ does not introduce new language syntax in Java). The transformation is derived based on our knowledge in RTSJ and our defined mappings from standard Java classes to RTSJ classes. The target Java Model′ is then used as an input for a model-to-text transformation, which is responsible to transform Java models back to Java source code.

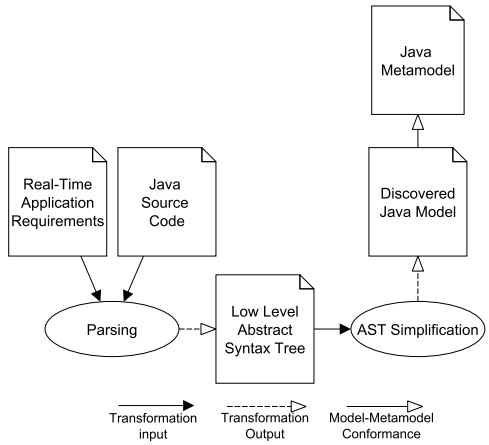

Fig. 3. Discovering Java a model from Java source code

With the proposed approach, applications developed originally in standard Java can be automatically converted to real-time applications based on RTSJ, and are directly executable on JamaicaVM. The whole conversion process is conducted without intervention of software developers, and hence, eliminates human-related erroneous factors. In addition, the proposed methodology removes the need of expertise in the real-time systems design and necessary knowledge of any targeted real-time programming specification. Consequently, the cost for real-time systems development can be significantly reduced with the high productivity brought by MDE. In the following sections, we will discuss the transformations involved in the approach individually.

4.1 Reverse Engineering Transformation (T2M)

The reverse engineering transformation is the very first transformation in the tool chain. Reverse engineering transformation is also normally referred to as *Model Discovery*, in the sense that a model is *discovered* from the source code. There are a number of available tools and approaches that are capable of performing this task. For example, JaMopp [23], Spoon [39] and MoDisco [9] are all feasible tools to perform reverse engineering from Java. It is to be noted that in this step, there is a strict requirement for model discovery in our proposed approach, that there shall be no information loss during the model discovery. This is typically due to the fact that the discovered model will be analysed, changed and then transformed back to the source code. If there is any information loss, the eventual transformed source code is not complete.

Our proposed reverse engineering transformation is presented in Figure 3. The Java source code and the real-time application requirements (we assume here that this would be a Java class with static fields) are firstly parsed into an Abstract Syntax Tree (AST), which is a very low-level representation model of the Java source code. The problem with ASTs is that they are difficult to navigate and analyse. Therefore, an AST simplification is performed to produce the discovered Java model that conforms to the Java Metamodel. The AST simplification is a reversible procedure, so that even if the discovered Java model is changed, the inverse of the AST simplification is still able to produce an AST that preserves all the original information (with changes applied to the AST).

4.2 Model Validation

With the discovered Java model, a model validation is performed to check whether the given application can meet its timing constraints defined in its real-time application requirements. The validation process first checks whether the given requirements are consistent with the input Java source code (e.g., whether threads' ids in the application are consistent with the ones given by the requirement). The real-time application requirements provide full thread releasing and scheduling information for each thread that needs to be transferred to real-time threads.

Then (assuming threads defined in the requirements are consistent with the source code), an analysis is performed to verify the timing properties of each real-time thread using the Response Time Analysis (RTA) discussed in [2]. For a given task τ_i in the targeted system, the worst-case response time R_i is then calculated by adding the task WCET C_i, the blocking time B_i and the interference time due to preemptions from higher-priority tasks I_i:

$$\begin{aligned} R_i &= C_i + B_i + I_i \\ &= C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \end{aligned} \tag{1}$$

where $\lceil \cdot \rceil$ denotes the ceiling function and $hp(i)$ returns the set of tasks that have higher priorities than τ_i's priority (denoted as $Pri(\tau_i)$ in the following equations).

In the presence of shared resources, C_i is further extended to include the time that τ_i spends on executing with each shared resources, as shown in (2).

$$C_i = WCET_i + \sum_{r^k \in F(\tau_i)} N_i^k c^k \tag{2}$$

where $WCET_i$ denotes the WCET of τ_i without accessing any shared resources, $F(\tau_i)$ gives a set of resources accessed by τ_i, c^k gives the WCET of a given resource k and N_i^k returns the number of access τ_i can issue to resource k in one release. It is to be noted that, as described in [49], the overhead cost by the real-time GC for allocating objects are included into the WCET of each thread, which should be pre-defined in the application requirements based on memory usage of each thread (i.e., keyword *new*).

The blocking time B_i indicates the time period that task τ_i is prevented from executing due to either non-preemptive sections from the underlying operating system or a low-priority thread that accesses a shared resource with a ceiling priority higher than $Pri(\tau_i)$.

$$B_i = max\{\hat{c_i}, \hat{b}\} \tag{3}$$

in which $\hat{c_i}$ denotes blocking due to resource access and $\hat{b}$ gives the longest non-preemptive section period in the underlying operating system. Finally, $\hat{c_i}$ is computed by (4) with PCP assumed. It is to be noted that the value of $\hat{b}$ depends on the operating system and underlying hardware, and should be measured and reported in the input application requirements.

$$\hat{c_i} = \max\{c^k | N_{lp}^k > 0 \wedge Pri(r^k) \geq Pri(\tau_i)\} \tag{4}$$

This equation finds all resources that are accessed by tasks with a lower priority but have a higher ceiling priority than $Pri(\tau_i)$, and returns the longest critical section among these resources as the worst-case blocking time for τ_i.

With the above analysis, the worst-case response time for each release of the application threads is bounded. If the system validation yields a schedulable system with given threads' scheduling parameters in the requirement, the proposed methodology processed to the next step, where it transfers the standard Java model to the real-time Java model based on the pre-generated metamodel.

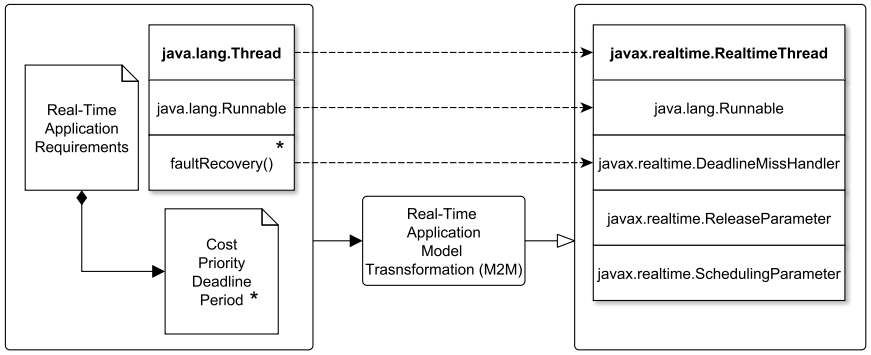

Fig. 4. Example transformation rule *Thread2RealtimeThread*

However, the current version of model validation heavily depends on system requirement for pre-measured computation cost of each thread and shared resource. In the future, mature worst-case execution time measuring techniques can be integrated into the proposed toolchain for a higher degree of automation.

4.3 RTSJ Model Transformation

After the RTA (model validation) test passes, in the next step, a model-to-model transformation (i.e., block *M2M* in Figure 2) is performed to migrate the standard Java model to a RTSJ Java model. It is to be noted that this transformation is endogenous in the sense that RTSJ does not introduce new Java abstract syntax, therefore the both the source model and the target model conform to the same Java Metamodel. This migration is performed based on a set of transformation rules, which specify the mapping from standard Java facilities to RTSJ facilities provided in package `javax.realtime`. In this section, we elaborate on two major Java to RTSJ transformations (i.e., threads and synchronisation) and then briefly describe the transformation rules to resolve RTSJ run-time environment dependencies.

4.3.1 Standard Threads to RTSJ Threads.

One major difference between standard Java and RTSJ is the schedulable entities (i.e., threads), where Java uses `java.lang.Thread` while RTSJ applications relies on `javax.realtime.RealtimeThread`. Figure 4 shows an example transformation rule (named *Thread2RealtimeThread*), which transforms a standard Java thread into a real-time thread.

On the left side of the figure is the source model of the transformation. The source model contains a number of standard Java threads that are extracted from the input source code. The transformation rule maps each standard Java thread to a real-time thread in RTSJ, as seen in the target model. In the source model, each thread is associated with a explicitly defined `java.lang.Runnable` object, which contains all functionality implementations that should be executed by this thread. This Runnable object will be passed directly into the `run()` method[9] of the real-time thread constructed during this transformation phase. In addition, each standard Java thread may also define an optional `faultRecovery()` method, which contains recovery operations to be performed in situations where the thread misses its deadline. The transformation rule transforms the code in the

[9]This Runnable object does not replace the Runnable of the real-time thread. It is passed into the real-time thread and executed by invoking its `run()` method for the execution of the logic implementation. The `run()` method of a real-time thread may contain extra implementation for realising its activation behaviours.

faultRecovery() method into RTSJ dedicated handlers, in this instance, the transformation creates a DeadlineMissHandler based on javax.realtime.AsyncEventHandler. If such a method is not provided, an immediate system shutdown is triggered in case of any deadline misses.

Then, the transformation rule creates RTSJ parameters such as ReleaseParameter and SchedulingParameter to apply timing constraints. As defined in the previous section, the source model also contains a set of real-time application requirements, which is defined in the form of a Java class with static fields. In real-time application requirements, a set of release and scheduling parameters specifying the execution eligibility and temporal proprieties of each thread should be defined by the users, as shown in Figure 4. Note that a period parameter is mandated for periodic threads, but should be omitted by those aperiodic threads to facilitate the identification of various activation pattern of real-time threads. Listing 3 provides examples of real-time application requirements for a periodic thread and an aperiodic thread, and IDs of their corresponding standard Java threads.

```
// Periodic thread pt1
public static final String pt1_jthread = "java_thread1";
public static final String pt1_id = "pt1";
public static final int pt1_period = 250;
public static final int pt1_cost = 200;
public static final int pt1_deadline = 250;
public static final int pt1_priority = 20;

// Aperiodic thread at1
public static final String at1_jthread = "java_thread2";
public static final String at1_id = "at1";
public static final int at1_cost = 30;
public static final int at1_deadline = 50;
public static final int at1_priority = 25;
```

Listing 3. Example application requirements of real-time threads in RTSJ

During the transformation, the priorities of the threads are constructed as javax.realime.PriorityParameter objects and other parameters (e.g., cost, deadline) are constructed as javax.realtime.ReleaseParameters objects, as shown in Figure 4. Depending on whether a given thread is associated with a period, the release parameter objects are further transformed into either PeriodicParameters or AperiodicParameters[10] objects provided in java.realtime.package. In addition, it is to be noted that the DeadlineMissHandler transformed from the pre-defined faultRecovery() method is also integrated into the ReleaseParameters object, as defined by the RTSJ specification.

With above real-time thread parameters and logic constructed, a standard Java thread can be transformed into a RTSJ thread by passing these parameter objects and the Runnable object into the construction method, where the Runnable object is called inside the run() method. For periodic threads, method waitForNextPeriod() is added into its run() method to achieve a periodic release behaviour. Listing 4 gives the implementation of the transformed periodic real-time thread pt1, where the parameters given in the application requirement (represented as the SystemSpec.java file) are assigned to the periodic thread, and the Runnable object of the corresponding standard Java thread is assigned to the generated PeriodicThread object. The helper method getThread() is pre-defined by the proposed methodology and is generated by the tool-chain during model

[10]Classes PeriodicParameters and AperiodicParameters are defined by RTSJ as the sub-classes of ReleaseParametersin package javax.realtime.

transformation. In addition, the object `deadMissHandler` is also generated during model transformation, and takes the `faultRecovery()` method defined in the input source code as its `Runnable` parameter.

Finally, note that as we assume the presence of a real-time GC, real-time threads are allowed to execute in Heap memory so that memory area parameters associated to the real-time threads are set as empty, which by default are executed in Heap by JamaicaVM. In addition, it is to be noted that we used a hybrid (i.e. imperative and declarative) transformation approach. The transformation rule in Figure 4 is one of the rules we define for the entire transformation. There are also rule dependencies, for example, we also define a *faultRecovery2DeadlineMissHandler* transformation rule, this rule should be called within our *Thread2RealtimeThread* transformation rule. This execution behaviour is typical for hybrid transformations and we recommend using the Epsilon Transformation Language [30] to write and execute the transformation. Finally, it is noted that we are not providing any concrete example of the source code of the standard Java application, as a standard Java thread can be implemented by various approaches and identified during the T2M transformation regardless of its actual implementation, where its `runnable()` object will be passed directly into the generated real-time thread for execution.

```
public class RTPeriodicThread extends RealtimeThread {
    public RTPeriodicThread(String name, int priority, int period, int cost, int deadline,
         Runnable logic, AsyncEventHandler deadlineMissHandler) {
             PriorityParameters priorityP = new PriorityParameters(priority)
             PeriodicParameters releaseP = new PeriodicParameters(clock.getTime(), new
                  RelativeTime(period, 0), new RelativeTime(cost, 0), new
                  RelativeTime(deadline, 0), null, deadlineMissHandler);
             super(priorityP, releaseP, null, null, null);
             this.setName(name);
    }
    @Override
    public void run() {
             while (!finished) {
                  logic.run();
                  waitForNextPeriod();
             }
    }
}
...
RTPeriodicThread pt1 = new RTPeriodicThread(SystemSpec.pt1_id, SystemSpec.pt1_priority,
     SystemSpec.pt1_period, SystemSpec.pt1_cost, SystemSpec.pt1_deadline,
    Utility.getThread(SystemSpec.pt1_jthread));
```

Listing 4. Code example of a transformed periodic thread

4.3.2 *Java Synchronisation to Real-Time Resource Control.*

Besides thread scheduling, another major difference between standard Java application and RTSJ is thread synchronisation, where in RTSJ each shared resource must be protected by proper resource sharing protocols (i.e., `javax.realtime.MonitorControl`) to ensure bounded resource accessing time for each resource access. As described in Section 3.3, the PCP is applied in the proposed toolchain for managing shared resources in transformed RTSJ appellations. This section describes the transformation rule *Lock2RealtimeLock* that transforms standard Java synchronisation to RTSJ

PCP facility[11]. In the current version of the proposed toolchain, we assume that each thread can only access one resource at a time.

To perform proper transformation, we define a set of rules towards thread synchronisation in the input source code and real-time application requirements, as described below.

- The user should be aware of all shared objects (i.e., ones that are read and written by more than one threads) and the threads that access those shared objects in the input Java source code.
- Each shared resource must be implemented as a class with the required operations implementing its methods properly protected by the standard Java synchronisation approach, i.e., via synchronised coding blocks and methods.
- The use of `wait()`, `notify()` and `notifyAll()` facilities in Standard Java are not allowed for thread synchronisation i.e., threads are not self suspended for accessing shared resources.
- For a given shared resource, say r^k, the user should provide its ceiling priority (i.e., the maximum priority of threads that access r^k) in the real-time application requirements.

In Listing 5, we provide an example of a valid input source code of a shared object class `SharedResource` and the associated application requirements conforming to the rules defined above.

```
// real-time application requirements
public static final String sr1_id = "sr1";
public static final int sr1_ceiling = 25;
...
// input source code
class SharedResource{
    String id;

    public synchronised void access(){
        critical_section;
    }
}

SharedResource sr1 = new SharedResource(sr1_id);
```

Listing 5. Application requirements and input source code of a shared resource in standard Java

With above rules defined, the standard Java synchronisation approach can be effectively transformed to their real-time resource sharing counterparts. First, in the source Java model generated in Section 4.1 (i.e., the model that extracts all objects in the input source code) , we are able to identify all shared objects (i.e., classes) and required operations by detecting the *synchronised* keyword. Then, for each object created based on these classes, its associated ceiling priority can be found in the application requirements. With the above information, RTSJ implementation can be generated by adding a `PriorityCeilingEmulation` instance to that object, as given in Listing 6.

The transformation first generates a `PriorityCeilingEmulation` instance for the shared object with its ceiling priority assigned based on the requirements. Then, the control policy for this shared resource is set to this PCP instance so that each thread that accesses this object will raise its priority, and later on restore its original priority once the lock is released. This is performed automatically by JamaicaVM, assuming the transformation is conducted successfully.

[11]The PCP in RTSJ is implemented by class `PriorityCeilingEmulation` in package `javax.realtime` by extending class `MonitorControl`.

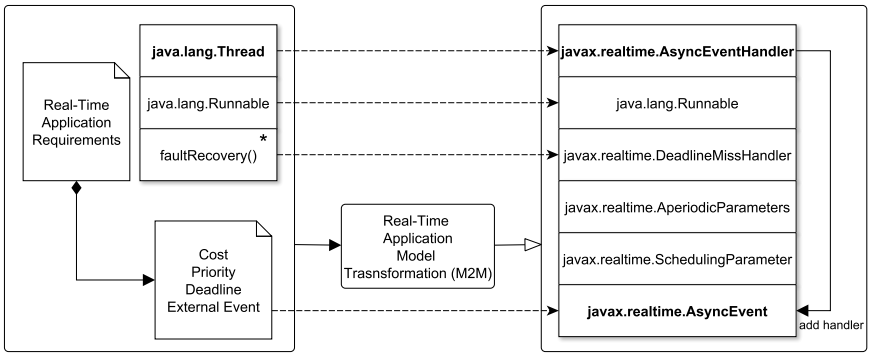

Fig. 5. Example transformation rule *Thread2AsyncEventHandler*

```
// Transformed RTSJ implementation
SharedResource sr1 = new SharedResource(sr1_id);
// With PCP enforced.
PriorityCeilingEmulation pcp = PriorityCeilingEmulation.instance(sr1_ceiling);
MonitorControl.setMonitorControl(sr1, pcp);
```

Listing 6. An example of RTSJ synchronisation with PCP applied

Finally, note that the priority ceiling priorities of shared objects must be correctly assigned in the real-time application requirements. Otherwise (e.g., the ceiling is lower than that of the accessing thread), a `CeilingViolationException` will be thrown by JamaicaVM.

4.3.3 Supporting Asynchronous Event Handling.
In most embedded systems, applications should be aware of environmental objects (e.g., sensors and controllers in robots, engines and vehicles), and then react to environmental changes for the assurance of the correctness of system execution. For example, in a hospital intensive care unit, the system must react if the "panic" button is pressed, and then send a paging information to the patient's duty doctor for immediate health care [58]. In such a system, the action "pressing the panic button" is an asynchronous event and can be triggered at any time from the external environment. The interactions between the application and environmental objects could be realised with either time-triggered or event triggered approaches. The time-triggered approach can be realised by periodically-released real-time threads in RTSJ e.g., a periodic thread that keeps monitoring the status of the "panic" button, and are supported by the proposed methodology in Sections 4.3.1 and 4.3.2. In contrast, the event-triggered programming provides a more flexible programming style, where threads are triggered by the occurrence of external events via interrupts. In addition, the event-triggered approach avoids additional overheads for maintaining many threads (e.g., using one thread for each environmental object), and thus, reduces the efforts for thread communication and synchronisation.

As described in Section 4.3.1, in RTSJ, Asynchronous Event Handlers (AEHs) are supported to handle situations where tasks miss their deadlines, which share the same philosophy as the Asynchronous Events (AEs) triggered by the external environment. Similarly, external events are also handled by AEHs in RTSJ specification. To support event-triggered programming in the proposed methodology, transformation rule *Thread2AsyncEventHandler* is defined to provide

automated transformation from standard Java threads to AsyncEventHandler objects in RTSJ, as illustrated by Figure 5.

For each external event, it is bind to an AsyncEvent object in RTSJ, which in turn, is associated with an AsyncEventHandler object. To transform the RTSJ event handling facility for one external event from the standard Java application, the handling thread (along with its Runnable object and the faultRecovery() method) should be provided in the input source code. In addition, the reference of the external event, the scheduling and release parameters for the transformed AsyncEventHandler object must be specified in the application requirement. Similar with the *Thread2RealtimeThread* transformation, the Runnable object and the faultRecovery() method of the standard Java thread are mapped to the Runnable object and the DeadlineMissHandler respectively in the generated AsyncEventHandler object. In addition, the scheduling and release parameters given in the application requirement are assigned to the generated AEH. Note that all AEHs are released aperiodically in the proposed methodology. Then, an AsyncEvent object is created to represent the external event via binding the event's identifier given by the application requirement. Finally, the AEH is registered to the AE so that it is executed each time the associated event is fired.

```
// Happening of External Event
public static final String ee1_id = "ee1";

// Asynchronous Event Handler
public static final String aeh1_jthread = "thread3";
public static final String aeh1_id = "aeh1";
public static final int aeh1_cost = 10;
public static final int aeh1_deadline = 250;
public static final int aeh1_priority = 10;

// Asynchronous Event
public static final String ae1_id = "ae1";
public static final String ae1_binding = ee1_id;
public static final String ae1_handling = aeh1_id;
```

Listing 7. Example real-time application requirements of an asynchronous event handling facility

Listing 7 provides an example application specification of asynchronous event handling for an external event, which contains the identifier of the external event, the identifier of the handling thread in the input source code, and parameters of the associated AE and its handler. For each asynchronous event, the references of the external objects and its handler are given for the binding. Note that as defined by RTSJ specification, the happening of an external event is modelled by a String object, which gives the handle of the external object.

```
public class RTAsyncEventHandler extends AsyncEventHandler {
    public RTAsyncEventHandler(String name, int priority, int cost, int deadline, Runnable
        logic, AsyncEventHandler deadlineMissHandler) {
            PriorityParameters priorityP = new PriorityParameters(priority);
            AperiodicParameters releaseP = new AperiodicParameters(new RelativeTime(cost,
                0), new RelativeTime(deadline, 0), deadlineMissHandler);
            super(priorityP, releaseP, null, null, null, false);
            this.setName(name);
    }
    @Override
    public void handleAsyncEvent() {
             logic.run();
```

```
    }
}
...
RTAsyncEventHandler aeh1 = new RTAsyncEventHandler(SystemSpec.aeh1_id,
     SystemSpec.aeh1_priority, SystemSpec.aeh1_cost, SystemSpec.aeh1_deadline,
     Utility.getThread(SystemSpec.aeh1_jthread));
RTAsyncEvent ae1 = new RTAsyncEvent(SystemSpec.ae1_id);
ae1.setHandler(Utility.getHandler(SystemSpec.aeh1_id));
ae1.bindTo(SystemSpec.ae1_binding);
```

Listing 8. Output implementation of the RTSJ asynchronous event handling facility

With the transformation rule *Thread2AsyncEventHandler* applied, the corresponding `AsynEvent` and `AsynEventHandler` objects conform to RTSJ can be generated, as given in Listing 8. The standard Java thread in the input source code is passed to the constructor of `aeh1` object, and its `run()` method is executed in the `handleAsyncEvent()` method of the `RTAsyncEventHandler` class. In addition, its name, scheduling and aperiodic release parameter are also assigned during the construction of the AEH objects, based on the given application requirement. Then, the `AsyncEvent` object is generated, which sets the `aeh1` object as its handler and is bind to the external event `ae1_binding`. After the binding, this asynchronous event handling facility is activated, where `aeh1` will be executed each time when the event occurs. Finally, as the JamacaVM and its real-time GC is applied, the objects of external environment happening (e.g., a sensor monitor object) are created in the Java Heap memory space. Similarly, their associated events and handlers are also created in this memory area, and thus, the memory-related parameters of the generated AEHs are set to be empty and the `noHeap` flag is set to false.

Theoretically, each AE can be associated with more than one AEHs and vice versa. However, for simplicity, the current version of the proposed methodology defines that one AE is designated to one AEH only, but allows that each AEH can be associated with more than one AEs (e.g., one deadline-miss handler can be associated with the deadline-miss events of multiple schedulable objects). When an external event is fired, its associated AE is stored into a queue and will be handled by a server thread. The AEs are ordered by the priories of their associated handlers. The server thread takes AEs from the queue individually according to the priority order, and executes their associated AEHs. The priority of the server thread is dynamic and reflects the priority of the executing AEH. As the current version of the proposed methodology aims at the uni-processor systems, this single server thread execution model is sufficient. With such a single server thread model, there is no need for explicit communication facility between the handlers as they simply read and write from shared objects one by one with no contention. However, as stated in [19], this execution model can raise the issue of unbounded priority inversion [58], where a low priority executing handler prevents the execution of a high priority queuing handler, and in turn, be preempted by a real-time thread with an intermediate priority. To bound the priority inversion, the proposed method defines that the priority of the server thread should be the maximum value between the priority of the handler it is currently executing and the priority of the handler at the front of the event handler queue, assuming the fixed-priority scheduling scheme.

Further, as described in [19], a multiple server threads execution model (with one or more handler queues) is more flexible and is desirable in multiprocessor real-time systems. However, using such a event handing model requires additional facilities for handler allocation and synchronisation. Extensions towards the asynchronous event handling model is a part of the future research topic given in Section 6, where the proposed methodology can be extended for multiprocessor real-time systems.

4.3.4 Supporting Asynchronous Transfer of Control.
The last major RTSJ facility supported by our proposed methodology is the Asynchronous Transfer of Control (ATC) facility, where the point of execution of a schedulable object (e.g., a real-time thread or an asynchronous event handier) can be changed by another schedulable object [58]. ATC is a common feature supported by many real-time programming language specifications (e.g., Ada), and provides an approach to enable immediate response of schedulable objects to certain conditions, such as recovery from a system error or execution mode changes. For instance, if a real-time thread detects a major system fault, it must inform other real-time threads quickly and safely so that immediate recovery actions can be performed collectively by these schedulable objects in a coordinated fashion.

In standard Java applications, asynchronous thread control is commonly realised by thread interruption via calling `interrupt()` method explicitly on the target thread. Then, an `InterruptedException` can be thrown by the target thread, and will be handled in the designated `catch` clause. However, this approach cannot guarantee immediate attention of the target threads when the interruption is fired. With standard Java exception handling model, interruptions can be delayed if the target threads are not blocked in interruptible methods (i.e., `join()`, `sleep()` and `wait()` methods). Therefore, a standard Java thread has to check for its interruption status periodically via the `isInterrupted()` method, and hence, can incur the delay of the interval of the checking period.

RTSJ achieves the ATC feature via extending the standard Java exception handling model and thread interruption facility, where the exception "Asynchronously Interrupted Exception" (AIE) is provided to interrupt real-time schedulable objects. In RTSJ, invocation to the `interrupt()` method of a schedulable object automatically throws the system generic AIE, rather than the `InterruptedException` in Standard Java. To support asynchronous interruption, the specification requires the that all methods that allow the delivery of an AIE (i.e., can be interrupted) must place this exception explicitly in their throw list. Such methods are termed as the asynchronously interruptible methods (i.e., AI-methods). When a schedulable object (e.g., a real-time thread) is interrupted (by calling `interrupt()` method) while executing in an AI-method, an AIE is generated by the underlying VM and is thrown immediately. In contrast, methods that do not throw AIE or with the `synchronized` keyword are termed as ACT-deferred methods. In such methods, an AIE is still generated but is held pending until the thread enters into a AI-method. By doing so, methods without ATC concern or access shared resources can still execute safely in the presence of an AIE.

However, as defined in RTSJ specification, AIE can only be handled by ATC-deferred methods. This is to prevent the situation where the handler is interrupted by another AIE. In addition, even if the AIE is caught and handled, it will still keep propagating until `clear()` method is called. This method tests whether the AIE is pending on the currently-executing schedulable object. If so, the AIE is set to non-pending and will stop propagation. Otherwise, the AIE can be left non-handled and be delivered to the calling method. With ATC applied, the output RTSJ application removes the need for polling the thread's interruption status periodically and can guarantee immediate response of the interrupted schedulable objects.

As the ATC facility is mainly supported by the underlying VM (e.g., the JamaicaVM), limited efforts are requirement to achieve the transformation from standard Java thread interruption to RTSJ ATC. Transformation rule *Interruption2ATC* is defined by the proposed methodology for this transformation. Firstly, during the T2M transformation phase (recall Figure 2), the interruption delivery methods and the handling method in the standard Java input source code can be identified via examining the `throws InterruptedException` clause and the `catch` clause respectively. Then, the `InterruptedException` is replaced by the `AsynchronouslyInterruptedException` in these methods during the M2M transformation phase. In addition, the `clear()` method is added into the

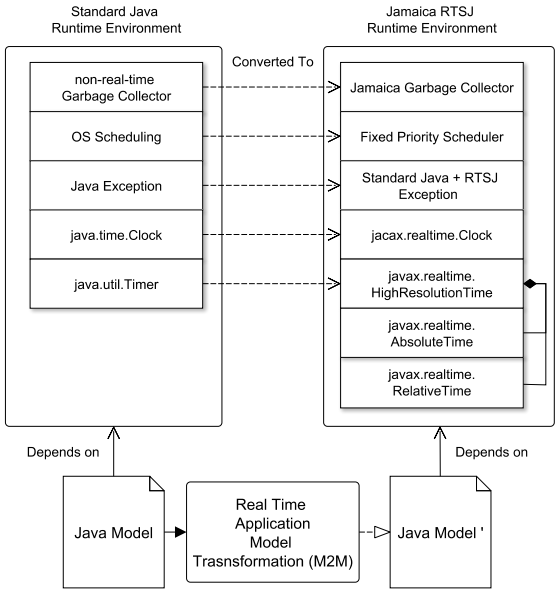

Fig. 6. Transformation rules to resolve run-time environment dependencies

`catch` clause of the handling methods to test whether the caught AIE is the one that is pending on the current method. If so, the handling code in the input source code will be executed. Otherwise, no actions are required and the AIE will keep propagating until it is caught by the pending method. This checking is done automatically by the underlying JamaicaVM.

Finally, note that in the current version of proposed methodology, a method that both throws and catches an AIE is not allowed and will results into an invalid Java Model during the Model Validation phase i.e., nested ATC model is not supported. Extensions towards supporting more flexible ATC model (e.g., multiple AIEs and nested ATC model) in the proposed Java to RTSJ automation methodology is listed as a possible future research topic in Section 6.

4.3.5 Transformation Rules for Run-Time Environment Dependencies.
Besides the transformation rules discussed above, RTSJ applications require a dedicated run-time Environment in order to be executed with real-time properties. Therefore, there is a need to execute a transformation to convert standard Java run-time Environment dependencies into RTSJ run-time Environment dependencies.

Figure 6 illustrates the transformation rule that converts these run-time environment dependencies. As shown in the figure, a typical standard Java run-time environment is equipped with a non-real-time garbage collector, a system clock with granularity in milliseconds, utility timers, standard Java exceptions. In addition, standard Java threads are mapped to native level threads and are scheduled by the underlying operating system.

In order for the output application with the RTSJ facilities generated in the above sections to execute successfully and to satisfy the application's timing requirements, a RTSJ run-time environment is required. The right side of Figure 6 shows an example of JamaicaVM-based RTSJ run-time environment. This JaimaicaJM RTSJ run-time is equipped with a dedicated real-time garbage collector running in the Heap memory, a real-time wall clock, finer-grained HighResolutionTime objects with granularity in nanoseconds and additional RTSJ related exceptions and a Fixed Priority

preemptive scheduler mechanisms. The mappings from Standard Java run-time environment to RTSJ run-time environment is drawn in Figure 6 using dashed lines.

Among the facilities considered in the targeted RTSJ run-time environment, the real-time garbage collector is enabled and the fixed priority scheduler is applied as default by JaimaicaVM. The standard Java clock is transformed to `javax.realtime.Clock` in `javax.realtime` package so that the invocations to obtain the current system time[12] is replaced by the method `Clock.getRealtimeClock().getTime()`. Further, as required by the RTSJ specification, time units in RTSJ should be modelled by `HighResolutionTime` as either a `AbsoluteTime` or a `RelativeTime` object, where the later two time units are sub-classes of the former. For instance, the temporal properties (e.g., period, cost and deadline) for a real-time thread will be generated as the `RelativeTime` objects before they are assigned to the construction method of `RealtimeThread`. Finally, additional exceptions introduced by Class RTSJ is generated into the output implementation where applicable. For instance, for each synchronised method, a `CeilingViolationException` exception should be thrown for illegal ceiling priority assignment. After this transformation is executed, the target model should have dependencies to RTSJ run-time resolved.

5 MODELLING SUPPORT OF RTSJ APPLICATIONS — A SYNTHESIS METHODOLOGY

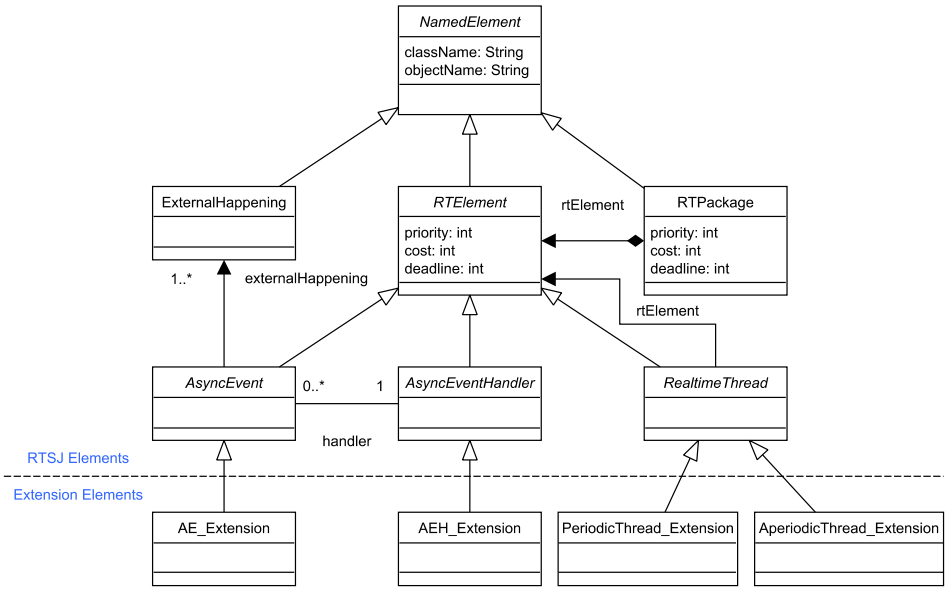

Fig. 7. A light weight metamodel for modelling with RTSJ

In previous sections, an automation methodology for standard Java to real-time Java application migration is discussed, which provides conversion from standard Java applications to real-time applications that are compliant to RTSJ. However, engineering scenarios do not always begin with creating a standard Java application. In certain cases (e.g., developing a new application), engineers typically have to start from scratch to create real-time Java applications i.e., without any reference implementation by the standard Java programming language. To ease the development, testing

[12]The method `System.currentTimeMillis()`.

and verification process of creating new RTSJ applications, a light-weight modelling language is proposed in this section, which allows engineers to create a RTSJ model that describes their real-time applications. Then, with the user-defined RTSJ application model in hand, model-to-text transformations are performed to generate structural Java source code (using RTSJ constructs) as the final output, to improve the efficiency of engineers throughout the entire development cycle and to eliminate human-related errors via automation.

Figure 7 shows the proposed light weight RTSJ metamodel. The root element is `NamedElement`, in which engineers can use `className` to name their class extension (e.g. extensions to AsyncEvent), and use `objectName` to name their objects. Meta-element `RTElement` directly inherits `NamedElement`, where users can describe the real time properties of their real-time objects (e.g. a real-time thread or an AEH) using scheduling and release proprieties `priority, cost and deadline`. Meta-element `RTPackage` maps to a Java package, it may contain RTSJ Classes. Meta-element `ExternalHappening` is used to capture external events that may be wrapped as an AsyncEvent. Abstract meta-element `AsyncEvent` is used to model an Asynchronous Event in RTSJ, it may refer to `ExternalHappening`(s). Abstract meta-element `AsyncEventHandler` is used to model an asynchronous event handler for Asynchronous Events in RTSJ. Note that an `AsyncEventHandler` may refer to a number of `AsyncEvent`, but an `AsyncEvent` must have one `AsyncEventHandler`. Meta-elements `AE_Extension` and `AEH_Extension` enables the users to create their own extensions to `AsyncEvent` and `AsyncEventHandler` respectively. Abstract meta-element `RealtimeThread` is used to capture a real-time thread object in RTSJ. The extensions of `RealtimeThread` (i.e., `RTPeriodicThread` and `RTAperiodicThread`) are provided for users to allow different real-time thread release models, as specified in RTSJ specification.

```
[%
var className = self.className;
var ePackageName = self.eContainer().name;
%]
package [%=ePackageName%];

import javax.realtime.AsyncEventHandler;
import javax.realtime.PriorityParameters;
import javax.realtime.AperiodicParameters;
...

public class [%=className%] extends AsyncEventHandler {

   public static final String priority = [%=self.priority%];
   public static final String cost = [%=self.cost%];
   public static final String deadline = [%=self.deadline%];
   public static final String name = [%=self.objectName%];
   protected Runnalbe logic = null;

   public [%=className%](Runnable logic, AyncEventHandler deadlineMissHandler)
   {
      PriorityParameters priority_parameters = new PriorityParameters(priority);
      AperiodicParameters release_parameters = new AperiodicParameters(new RelativeTime(cost, 0),
                                 new RelativeTime(deadline, 0),
                                 deadlineMissHandler);
      super(priority_parameters, release_parameters, null, null, null, false);
      this.setName(name);
      this.logic = logic;
   }
```

```
  public [%=className%](String name, int priority, int cost, int deadline, Runnable logic,
       AyncEventHandler deadlineMissHandler)
  {
     PriorityParameters priority_parameters = new PriorityParameters(priority);
     AperiodicParameters release_parameters = new AperiodicParameters(new RelativeTime(cost, 0),
                                  new RelativeTime(deadline, 0),
                                  deadlineMissHandler);
     super(priority_parameters, release_parameters, null, null, null, false);
     this.setName(name);
     this.logic = logic;
  }

  @Override
  protected void handleAsyncEvent() {
     logic.run();
  }
}
```

Listing 9. Example Model-to-Text transformation written in EGL to generate an extension class of AsyncEventHandler

With the RTSJ models constructed, model-to-text transformations can be executed on them. Listing 9 shows a model-to-text transformation script written in Epsilon Generation Language (EGL) [42] for generating extension classes of `AsyncEventHandler`. For model-to-text generations, there are static sections and dynamic sections, dynamic sections in EGL are enclosed with square brackets (i.e. [and]). As it can be seen in Listing 9, dynamic sections uses information from the model to generate extension classes of `AsyncEventHandler`, with most static sections automatically generated, human error factors can be eliminated. It is to be noted that the model-to-text transformations provided in our approach only generate template-like Java classes, the developers typically need to write their algorithms and logic in their defined `Runnable` instances.

6 OPEN CHALLENGES AND FURTHER RESEARCH DIRECTIONS

Plenty of open questions and research opportunities are introduced by this work. In this section, we discuss some of the challenges and point towards selected future research directions.

First, the current version of the proposed toolchain assumes the presence of a real-time garbage collector (e.g., the one supported by JamaicaVM), which allows the execution of real-time threads in *Heap* memory. However, in situations where a real-time GC is not available, an explicit memory management model must be constructed by `ScpoedMemory` to guarantee temporal requirements of real-time Java applications, as executing in Heap memory will suffer from unpredictable interference of standard Java garbage collector. One major challenge of this Java to RTSJ automation approach is to provide a generic memory management model that suits all types of RTSJ applications. The memory management model in RTSJ is highly specific to the application characteristics (especially, the correlation of those real-time threads) and is difficult to generate based merely on the knowledge from the input source code. In addition, a future version of the proposed toolchain may include the measurement (or analysis) of certain task execution parameters (e.g., WCET) in the automation process, towards a higher degree of automation.

A possible workaround is to enforce an SCJ (Safety-Critical Java)-like programming model [13, 46], which imposes restrictions towards the application structure but is sufficient to provide the required functionalities. In the SCJ, threads are grouped into *missions*, which are executed by one or more *mission sequencers* (i.e., missions can be executed concurrently). This programming model conforms to a specific memory management framework [45]. In the SCJ, each mission has its own memory

block and each thread in that mission is also assigned with a private memory area, building upon the memory block of the associated mission. Once a mission is finished (i.e., all its threads are signalled to be terminated), its associated memory block (and subsequently, memory areas of its threads) will be reclaimed during a mission *cleanUp* phase. However, applying this memory model in the proposed methodology requires extra information describing the correlation between those real-time threads in order to allocate them correctly into each individual group for memory allocation.

Second, as an initial attempt on the topic, we have targeted at a simple and widely-applied uniprocessor environment and focused mainly on the functionality of the proposed toolchain. There is a trend that most of the existing real-time programming specifications are extended to support multiprocessor and distributed systems [62]. The proposed approach can also be extended to support multiprocessor features with multiprocessor scheduling policies, resource sharing techniques and multiple server thread asynchronous event handling model taken into account [60, 61]. In addition, as the application scenarios of real-time systems become more sophisticated, supporting complex system semantics (e.g., in the presence of release jitters, shared resources or nested AIEs) is also desirable and should be investigated.

In addition, as illustrated in Figure 2, there is an open question to be answered when the given applications are found unschedulable after model validation. One possible solution would be the reconfiguration of system scheduling parameters to achieve better schedulability (i.e., transferring systems that are deemed unschedulable into feasible real-time systems). Such reconfiguration is worthwhile especially for complex systems (e.g., multiprocessor systems with shared resources), where optimal scheduling solutions may not be available. In such cases, a search-based algorithm could be applied for searching threads' parameters and feasible resource sharing protocols that can achieve a schedulable system [59]. In addition, further improvement can be made towards other perspectives of real-time systems, such as sustainability and robustness in the presence of additional interference.

From the programming language perspective, the proposed automated toolchain can be generalised to support different programming languages (e.g., C/C++ and Ada) and their real-time, safety-critical and high-integrity extension profiles (e.g., MISRA C/C++ [22, 53] and Spark Ada [6]). Such efforts are worthwhile as they remove the restriction on the usage of a specific programming language (and its extensions) in the proposed automated toolchain and provide solutions towards those major programming languages in embedded systems.

From the model-driven perspective, for those programming languages where reverse engineering facilities may not be available (e.g., C and Ada), modelling real-time systems from system specification directly and then generating implementation via code generation facilities would be desirable. There are several modelling languages which are capable of modelling real-time systems, e.g., the Architectural Analysis and Design Language (AADL), the Unified Modelling Language (UML), the Systems Modelling Language (SysML), the Modelling and Analysis of Real-Time Embedded Systems (MARTE) UML profile, and the AADL for UML profile are all feasible languages for modelling real-time systems.

However, there are shortcomings in these languages discussed above. AADL is not an open modelling language, and there is a lack of modelling capabilities for the system behaviour. UML is a general modelling language. However, it lacks the formalism needed in modelling of the real-time systems. SysML shares the same problem as UML. MARTE provides extensive modelling capabilities, which leads to the complexity of the language itself. Consequently, a MARTE model could get complex quickly, leading to complex models and diagrams which are hard to manage. A new modelling language is therefore needed for the real-time systems community to address the

above shortcomings. With this modelling language, we could generate the real-time applications in programming languages such as Java, Ada and C.

7 CONCLUSION

In this paper, two model-based development methodologies have been proposed, which automatically generate RTSJ-compliant implementations through either transforming the existing time-sharing standard Java applications or modelling customised RTSJ applications using a light-weight modelling language from scratch. The proposed methodologies ease the development of real-time systems by allowing software engineers to construct real-time Java applications without necessary knowledge of the RTSJ programming specification. The proposed methodologies provide real-time system development solutions that reduce software development cost, increase productivity and eliminate human-related errors by automating the entire implementation process.

In particular, the Java to RTSJ transformation methodology is favourable to those organisations with a need to re-develop their products to possess real-time features while the RTSJ modelling approach is valuable when no standard Java reference implementation is available. The complete standard Java to RTSJ conversion automation architecture is presented, with the required actions in each transformation phase described in detail. Transformation rules are also presented for generating major RTSJ facilities and the RTSJ run-time environment based on the JamaicaVM with the given inputs. In addition, the RTSJ application modelling approach is illustrated and explained, with output implementation example provided using model-to-text transmissions.

The proposed methodologies raise plenty of research questions and possible research directions, which can be investigated together by the embedded systems, programming languages as well as MDE communities. They have been discussed with motivation and preliminary approaches. In future, we aim to provide a complete and fully functional Java to RTSJ transformation toolchain and RTSJ modelling tool-kit to fully exploit the potentials of Model-Driven Engineering for real-time programming using RTSJ, and to evaluate the efficacy of the proposed RTSJ development automation methods.

REFERENCES

[1] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. 2007. A real-time Java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 1 (2007), 5.

[2] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal* 8, 5 (1993), 284–292.

[3] Neil C Audsley, Yu Chan, Ian Gray, and Andy J Wellings. 2014. Real-Time Big Data: the JUNIPER Approach. (2014).

[4] Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbruster, Edward Pla, and David Holmes. 2006. A real-time java virtual machine for avionics-an experience report. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE, 384–396.

[5] Konstantinos Barmpis, Dimitrios Kolovos, and Justin Hingorani. 2018. Towards a framework for writing executable natural language rules. In *European Conference on Modelling Foundations and Applications*. Springer, 251–263.

[6] John Barnes. 1997. *High integrity Ada: the SPARK approach*. Vol. 189. Addison-Wesley Reading.

[7] Jean Bézivin. 2005. On the unification power of models. *Software & Systems Modeling* 4, 2 (2005), 171–188.

[8] Gregory Bollella and James Gosling. 2000. The real-time specification for Java. *Computer* 33, 6 (2000), 47–54.

[9] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. Modisco: A model driven reverse engineering framework. *Information and Software Technology* 56, 8 (2014), 1012–1032.

[10] Alan Burns and Andy Wellings. 2016. *Analysable Real-Time Systems: Programmed in Ada.* CreateSpace Independent Publishing Platform.

[11] Alan Burns and Andrew J Wellings. 2001. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education.

[12] Alan Burns, Andy J Wellings, and Fengxiang Zhang. 2009. Combining EDF and FP scheduling: Analysis and implementation in Ada 2005. In *International Conference on Reliable Software Technologies*. Springer, 119–133.

[13] Ana Cavalcanti, Alvaro Miyazawa, Andy Wellings, Jim Woodcock, and Shuai Zhao. 2017. *Java in the Safety-Critical Domain.* Springer International Publishing, Cham, 110–150. https://doi.org/10.1007/978-3-319-56841-6_4
[14] Wanli Chang and Samarjit Chakraborty. 2016. Resource-aware automotive control systems design: A cyber-physical systems approach. *Foundations and Trends in Electronic Design Automation* 10, 4 (2016), 249–369.
[15] Wanli Chang, Dip Goswami, Samarjit Chakraborty, and Arne Hamann. 2018. OS-aware automotive controller design using non-uniform sampling. *ACM Transactions on Cyber-Physical Systems* 2, 4 (2018), 26.
[16] Wanli Chang, Dip Goswami, Samarjit Chakraborty, Lei Ju, Chun Xue, and Sidharta Andalam. 2017. Memory-aware embedded control systems design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 4 (2017), 586–599.
[17] Wanli Chang, Shuai Zhao, Ran Wei, Andy Wellings, and Alan Burns. 2019. From Java to real-time Java: a model-driven methodology with automated toolchain. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 123–134.
[18] Robert I. Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *Acm Computing Surveys* 43, 4 (2011), 1–44.
[19] Peter Dibble. 2002. *Real-time Java platform programming.* Prentice Hall Professional.
[20] Ian Gray, Neil Cameron Audsley, Jamie Garside, Yu Chan, and Andrew John Wellings. 2015. FPGA-based acceleration for Real-Time Big Data Systems. In *9th HiPEAC workshop on Reconfigurable Computing*.
[21] Ian Gray, Yu Chan, Jamie Garside, Neil C. Audsley, and Andy J. Wellings. 2015. FPGA-based hardware acceleration for Real-Time Big Data systems.
[22] Les Hatton. 2004. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology* 46, 7 (2004), 465–472.
[23] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. 2009. Closing the gap between modelling and java. In *International Conference on Software Language Engineering*. Springer, 374–383.
[24] Thomas Henties, James J Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. 2009. Java for safety-critical applications. In *2nd international workshop on the certification of safety-critical software controlled systems (SafeCert 2009)*.
[25] Erik Yu-Shing Hu, Eric Jenn, Nicolas Valot, and Alejandro Alonso. 2006. Safety critical applications and hard real-time profile for Java: a case study in avionics. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*. ACM, 125–134.
[26] Ari Jaaksi. 2002. Developing mobile browsers in a product line. *IEEE software* 19, 4 (2002), 73–80.
[27] A Juan, Jorge Garrido, Juan Zamorano, and Alejandro Alonso. 2014. Model-driven design of real-time software for an experimental satellite. *IFAC Proceedings Volumes* 47, 3 (2014), 1592–1598.
[28] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. 2009. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*.
[29] Timothy Patrick Kelly. 1999. *Arguing safety: a systematic approach to managing safety cases.* Ph.D. Dissertation. University of York York, UK.
[30] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*. Springer, 46–60.
[31] Dimitrios S Kolovos, Ran Wei, and Konstantinos Barmpis. 2013. An approach for efficient querying of large relational datasets with ocl-based languages. In *XM 2013–Extreme Modeling Workshop*. 48.
[32] Björsn Konieczek, Michael Rethfeldt, Frank Golatowski, and Dirk Timmermann. 2015. Real-time communication for the internet of things using jcoap. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, 134–141.
[33] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. 2009. Papyrus UML: an open source toolset for MDA. In *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. Citeseer, 1–4.
[34] Shaoshan Liu, Jie Tang, Chao Wang, Quan Wang, and Jean-Luc Gaudiot. 2017. Implementing a Cloud Platform for Autonomous Driving. *arXiv preprint arXiv:1704.02696* (2017).
[35] Shaoshan Liu, Jie Tang, Chao Wang, Quan Wang, and Jean-Luc Gaudiot. 2017. A unified cloud platform for autonomous driving. *Computer* 50, 12 (2017), 42–49.
[36] No Magic. 2007. MagicDraw. (2007).
[37] HaiTao Mei, Ian Gray, and Andy Wellings. 2016. Real-Time stream processing in java. In *Ada-Europe International Conference on Reliable Software Technologies*. Springer, 44–57.
[38] Parastoo Mohagheghi and Vegard Dehlen. 2008. Where is the proof?-A review of experiences from applying MDE in industry. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 432–443.
[39] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015),

1155–1179. https://doi.org/10.1002/spe.2346
[40] Ben Potter, David Till, and Jane Sinclair. 1996. *An introduction to formal specification and Z.* Prentice Hall PTR.
[41] Ragunathan Rajkumar. 2012. *Synchronization in real-time systems: a priority inheritance approach.* Vol. 151. Springer Science & Business Media.
[42] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. 2008. The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 1–16.
[43] RTEMS. Accessed: 21-02-2020. http://www.rtems.org/
[44] Douglas C Schmidt. 2006. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-* 39, 2 (2006), 25.
[45] Martin Schoeberl, Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Stephan E Korsholm, Anders P Ravn, Juan Ricardo Rios Rivas, Tórur Biskopstø Strøm, Hans Søndergaard, Andy Wellings, and Shuai Zhao. 2017. Safety-critical Java for embedded systems. *Concurrency and Computation: Practice and Experience* 29, 22 (2017), e3963.
[46] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P Ravn. 2007. A profile for safety critical java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*. IEEE, 94–101.
[47] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. 39, 9 (1990).
[48] Fridtjof Siebert. 2007. Realtime garbage collection in the JamaicaVM 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. Citeseer, 94–103.
[49] Fridtjof Siebert. 2010. Concurrent, parallel, real-time garbage-collection. In *ACM Sigplan Notices*, Vol. 45. ACM, 11–20.
[50] Rashmi P Sonar and Rani S Lande. 2018. Javolution-Solution for Real Time Embedded System. In *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*. IEEE, 1–10.
[51] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework.* Pearson Education.
[52] Sparx Systems. 2012. Sparx Systems SysML. https://sparxsystems.com/. (2012). [Accessed: 23/03/2020].
[53] Chris Tapp. 2008. An introduction to MISRA C++. *SAE international journal of passenger cars-electronic and electrical systems* 1, 2008-01-0664 (2008), 265–268.
[54] Kleanthis Thramboulidis. 2007. IEC 61499 in factory automation. In *Advances in Computer, Information, and Systems Sciences, and Engineering*. Springer, 115–124.
[55] Kleanthis Thramboulidis and Alkiviadis Zoupas. 2005. Real-time Java in control and automation: a model driven development approach. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, Vol. 1. IEEE, 8–pp.
[56] Christian Wawersich, Michael Stilkerich, and Wolfgang Schröder-Preikschat. 2007. An OSEK/VDX-based multi-JVM for automotive appliances. In *Embedded System Design: Topics, Techniques and Trends*. Springer, 85–96.
[57] Ran Wei, Tim P Kelly, Xiaotian Dai, Shuai Zhao, and Richard Hawkins. 2019. Model based system assurance using the structured assurance case metamodel. *Journal of Systems and Software* 154 (2019), 211–233.
[58] Andrew J Wellings. 2004. *Concurrent and real-time programming in Java.* John Wiley New York.
[59] Shuai Zhao. 2018. *A FIFO Spin-based Resource Control Framework for Symmetric Multiprocessing.* Ph.D. Dissertation. University of York.
[60] Shuai Zhao, Jorge Garrido, Alan Burns, and Andy Wellings. 2017. New schedulability analysis for MrsP. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1–10.
[61] Shuai Zhao, Jorge Garrido, Ran Wei, Alan Burns, Andy Wellings, and A Juan. 2020. A complete run-time overhead-aware schedulability analysis for MrsP under nested resources. *Journal of Systems and Software* 159 (2020), 110449.
[62] Shuai Zhao, Andy Wellings, and Stephan Erbs Korsholm. 2015. Supporting multiprocessors in the ICECAP safety-critical java run-time environment. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, 1.
[63] Athanasios Zolotas, Ran Wei, Simos Gerasimou, Horacio Hoyos Rodriguez, Dimitrios S. Kolovos, and Richard F. Paige. 2018. Towards Automatic Generation of UML Profile Graphical Editors for Papyrus. In *Modelling Foundations and Applications*, Alfonso Pierantonio and Salvador Trujillo (Eds.). Springer International Publishing, Cham, 12–27.